phy++ v1.0

Contents

1 Introduction **4**

2 Core library **5**
2.1 Overview 5
2.1.1 Operator overloading 6
2.1.2 Multi-dimensional indexing 7
2.1.3 Array indexing 8
2.2 The vector class 9
2.2.1 Member variables 10
2.2.2 Constructors and assignment 10
2.2.3 Member functions 10
2.2.4 Indexing 10
2.2.5 Operators 10
2.3 The view class 10
2.3.1 Member functions 10
2.4 Metaprogramming traits 11

3 Support libraries **12**
3.1 Parsing command line arguments 13
3.2 File input/output 13
3.3 String manipulation 13
3.4 Printing to the terminal 13
3.5 Measuring time 13
3.6 Math and generic algorithms 13
3.7 Parallel execution 13
3.8 FITS input/output 13
3.9 Image processing 13
3.10 Astrophysics 13

3.10.1 Catalog management 13
3.10.2 Position cross-matching 13
3.10.3 Image stacking 13
3.10.4 Template fitting 13

4 Function index **14**

Chapter 1

Introduction

Chapter 2

Core library

2.1 Overview

At the core of the phy++ library is the *vector* class. This is basically an enhanced `std::vector`[1], and it therefore shares most of its features and strengths. In particular, a vector can contain zero, one, or as many elements as your computer can handle. Its size is defined at *run-time*, meaning that its content can vary depending on user input, but also that a vector can change its total number of elements at any time. These elements are stored contiguously in memory, which provides optimal performances in most situations. Lastly, a vector is an homogeneous container, meaning that a given vector can only contain a single type of elements (e.g., `int` or `float`, but not both at the same time).

Like most advanced C++ libraries, phy++ is essentially *template* based. This means that most of the code is written to work for *any* type `T`, e.g., `int`, `float`, `std::string`, or whatever you need. However, while templates are a fantastic tool for library writers, they can easily become a burden for the *user* of the library. The good thing is, since phy++ is a numerical analysis library, we know in advance what types will most often be stored inside the vectors. So for this reason, to reduce typing and enhance readability, we introduce type aliases for the most commonly used vector types:

- `vec1f`: vector of `float`,
- `vec1d`: vector of `double`,
- `vec1i`: vector of `int` (precisely, `int_t = std::ptrdiff_t`),
- `vec1u`: vector of `unsigned int` (precisely, `uint_t = std::size_t`),

[1] In fact, `std::vector` is used to implement the phy++ vectors internally.

- `vec1b`: vector of `bool`,
- `vec1s`: vector of `std::string`,
- `vec1c`: vector of `char`.

On top of the `std::vector` interface, the phy++ vector adds some extra functionalities. The most important ones are operator overloading, multi-dimensional indexing, and array indexing.

2.1.1 Operator overloading

The only thing you can do to operate on all the elements of an `std::vector` is to iterate over these elements explicitly, either using a C++11 range-based loop, or using indices:

```
// Goal: multiply all elements by two.
std::vector<float> v = {1,2,3,4};

// Either using a range-based loop,
for (float& x : v) {
    x *= 2;
}

// ... or an index-based loop.
for (std::size_t i = 0; i < v.size(); ++i) {
    v[i] *= 2;
}
```

While this is fairly readable (especially the first version), it is still not very concise and expressive. In phy++, we have *overloaded* the usual mathematical operators on our vector type, meaning that it is possible to write the above code in a much simpler way:

```
// Using phy++ vector.
vec1f w = {1,2,3,4};
w *= 2; // {2,4,6,8}
```

Not only this, but we can perform operations on a pair of vectors in the same way:

```
// Goal: sum the content of the two vectors.
vec1f x = {1,2,3,4}, y = {4,3,2,1};
vec1f z = x + y; // {5,5,5,5}
```

Warning

The only issue with operator overloading concerns the hat operator `^`. In most languages, this operator is used for exponentiation, e.g., `4^2 == 16`. However, in C++ this value is actually `6`, because the hat operator is the binary XOR (exclusive-or). Even worse, the hat operator in C++ does not have the same *precedence* as in regular mathematics: it has a lower priority than any other mathematical operator. Both these reasons make it unwise to overload the hat operator. In order to perform exponentiation, you will have to use a dedicated function such as `pow(4,2)` (see **??**).

2.1.2 Multi-dimensional indexing

The standard `std::vector` is a purely linear container: one can access its elements using `v[i]`, with `i` ranging from `0` up to `std::vector::size()-1` (included). However, the phy++ vector allows N-dimensional indexing, i.e., using a group of indices to identify one element. For example, this is particularly useful to work on images, which are essentially 2-dimensional objects where one identifies a given pixel by its coordinates `x` and `y`. The natural syntax for this indexing would be to write `img[x,y]`. This syntax is valid C++ code, but unfortunately will not do what you expect[2] and there is no sane way around it. The alternative we chose here is to write instead `img(x,y)`. While it is not as semantically clear as using brackets, it has the nice advantage of being compatible with the IDL syntax.

The multi-dimensional nature of a vector is determined at *compile time*, i.e., it cannot be changed after the vector is declared. By default, a vector is mono-dimensional. To use the above feature, one needs to specify the number of needed dimensions in the type of the vector. For example, a 2D image of `float` will be declared as `vec2f`. These type aliases are provided for dimensions up to 6. Here is an example of manipulation of a 2D matrix:

```
// Create a simple matrix.
vec2f m = {{1,2,3}, {4,5,6}, {7,8,9}};

// Index ordering is similar to C arrays: the last index
// is contiguous in memory. Note that this is *opposite*
// to the IDL convention.
m(0,0); // 1
m(0,1); // 2
```

[2] This will call the *comma* operator, which evaluates both elements `x` and `y` and returns the last one, i.e., `y`. So this code actually means `img[y]`. With proper configuration, most compiler will warn about this though, since in this context `x` is a useless statement, so you should be safe should you make this mistake.

```
m(1,0); // 4

// It is still possible to access elements as if in a
// "flat" vector
m[0]; // 1
m[1]; // 2
m[3]; // 4
```

Advanced

If for some reason you need to use more than 6 dimensions, or if you need to declare a vector of some type which is not covered above, you can always fall back to the full template syntax:

```
using vec12f = vec_t<12,float>;
using vec3cx = vec_t<3,std::complex>;
```

The hard limit on the number of dimensions will then depend on your compiler. The C++ standard does not guarantee anything, but you should be able to go as high as 256 on all major compilers. Beyond this, you should probably see a doctor first.
As for the types allowed inside the vector, there is no explicit restriction. However, some features may not be available depending on your type, and you will have to enable these yourself (operator overloading, in particular). Lastly, the phy++ vector shares the same restrictions as the `std::vector` regarding the *copyable* and *movable* capabilities of the stored type.

2.1.3 Array indexing

We have seen that, instead of accessing each element individually, we can use operator overloading to perform simple operations on all the elements of the vector at once: `w *= 2`. We can also, like with `std::vector`, modify each element individually, knowing their indices: `w[2] *= 2`. One last important feature allowed by the phy++ vector is array indexing, which allows us to create a *view* inside an array. Each element of the view is actually a reference to another element in the original array, and modifying the elements of the view actually modifies the values in the original array.

```
// Create a simple vector.
vec1f w = {1,2,3,4,5,6};

// We want to "view" only the second, the third and the
```

```
// fifth elements. So we first create a vector containing
// the corresponding indices.
vec1u id = {1,2,4};

// Then we create the view.
// Note the usage of "auto" there. The type of the view is
// complex, and it is better not to worry about it.
auto v = w[id];

// Now we can modify these elements very simply, as if they
// were part of a real vector.
v;          // {2,3,5}
v *= 2;     // {4,6,10}
w;          // {1,4,6,4,10,6}
w[1] = 99; // {1,99,6,4,10,6}
v;          // {99,6,10}
```

Warning

It is important to note that, since a view keeps *references* to the elements of the original vector, its lifetime must not exceed that of the original vector. Else, it will contain *dangling* references that point to unused memory, and this should be avoided at all cost. In fact, views are not meant to be stored into named variables like in the above example. Most of the time, one will use them as temporary variables, e.g.:

```
vec1f w = {1,2,3,4,5,6};
vec1u id = {1,2,4};
w[id] *= 2; // {1,4,6,4,10,6}
```

2.2 The vector class

The full type of the vector class is

```
template<std::size_t Dimension, typename ElementType>
struct vec_t<Dimension,ElementType>;
```

In the rest of this document, `Dimension` will usually just be called `D` and `ElementType` will be shortened to `T`.

2.2.1 Member variables

2.2.2 Constructors and assignment

2.2.3 Member functions

2.2.4 Indexing

2.2.5 Operators

2.3 The view class

The full type of the view class is

```
template<std::size_t Dimension, typename ElementType>
struct vec_t<Dimension,ElementType*>;
//              note the asterisk ^
```

The public interface of the view class is very similar to that of the normal vector, and we will not repeat it here. There are some important differences though, which are inherent to the goal of this class. In particular, there is no available constructor (you do now create a view yourself, you ask for it from an existing vector that will create it for you), and the `resize()` function is not available. Lastly, the view provides a new member function called `concretize()`, which is described below.

Thanks to their strong similarity, we will not distinguish in the other sections between vectors and views, and will consider views as just another kind of vectors. Indeed, the interface of these two classes has been designed for views to be completely interchangeable with vectors, and vice versa, so that the code of any given function is written once and is valid for both types.

2.3.1 Member functions

concretize

Signature: `vec_t<D,T> vec_t<D,T*>::concretize() const`

This function creates a new vector out of the elements of this view. The returned vector is completely independent from this view, or the original vector this view is currently pointing to. This function is mostly useful when writing generic functions. Indeed, views are implicitly convertible to normal vectors on assignment (see 2.2.2).

2.4 Metaprogramming traits

Warning

This whole section is more advanced than the rest. It is describing the sets of helper types and metaprogramming functions that one can use to write new functions. It is intended to be followed by readers already familiar with the `phy++` library.

Chapter 3

Support libraries

3.1 Parsing command line arguments

3.2 File input/output

3.3 String manipulation

3.4 Printing to the terminal

3.5 Measuring time

3.6 Math and generic algorithms

3.7 Parallel execution

3.8 FITS input/output

3.9 Image processing

3.10 Astrophysics

3.10.1 Catalog management

3.10.2 Position cross-matching

3.10.3 Image stacking

3.10.4 Template fitting

Chapter 4

Function index